

Паттерны организации консистентности(согласованности) и обмена данными

Рассмотрим на простом примере интернет-магазина. В данной системе у нас есть три микросервиса:

- 1. Сервис управления заказами
- 2. Сервис инвентаря (склада)
- 3. Сервис уведомлений

Сценарий: Покупка товара в интернет-магазине

Клиент хочет купить товар. Покупка включает в себя выбор товара, добавление его в корзину и оформление заказа, который, в свою очередь, включает в себя списание товара со склада и отправку уведомления клиенту о статусе заказа.

- **Проблема 1: Нам нужно сначала зарезервировать товар на складе, затем создать заказ в системе управления заказами, и после успешного завершения всех этих шагов отправить уведомление клиенту (то есть это некий аналог транзакции в монолите ACID). Нам поможет Паттерн Saga**

Saga — это паттерн проектирования, который позволяет обеспечить согласованность данных в распределенных системах. В контексте микросервисов это значит координировать несколько операций (локальных транзакций), каждая из которых выполняется в своём микросервисе.

Как это реализуется в монолите

В монолитных приложениях для обеспечения консистентности часто используется двухфазный коммит (2PC) или другие ACID-транзакции. Эти механизмы не применимы или неэффективны в распределенных системах из-за сетевых задержек, различий в хранении данных и других факторов.

Зачем этот паттерн в микросервисах

В микросервисах существует необходимость в обеспечении консистентности данных между разными службами, которые могут быть распределены по разным серверам или даже географически разнесены. Saga помогает решить эту проблему, обеспечивая организованный и управляемый способ выполнения последовательности операций через разные микросервисы.

Как он реализуется

Есть два способа реализации - оркестрация или хореография.

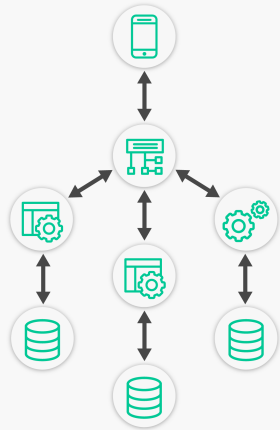
Оркестрация:

1. Сервис управления заказами ("Оркестратор") начинает Saga, отправляя команду на резервирование товара в сервис инвентаря.
2. Сервис инвентаря резервирует товар и возвращает ответ оркестратору.
3. Оркестратор создает заказ в своей базе данных.
4. Оркестратор отправляет команду на отправку уведомления в сервис уведомлений.
5. Уведомление отправлено, Saga завершена успешно.

Хореография:

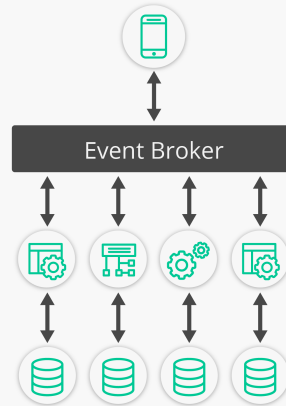
1. Сервис управления заказами начинает Saga, создавая заказ и публикуя событие "Заказ создан".
2. Сервис инвентаря подписан на это событие. Он резервирует товар и публикует событие "Товар зарезервирован".
3. Сервис уведомлений подписан на событие "Товар зарезервирован" и отправляет уведомление пользователю.
4. Saga завершена успешно.

Orchestration



VS

Choreography



Что лучше из подходов?

Оркестрация:

1. **Централизованное управление:** Оркестрация обычно подразумевает наличие центрального "дирижера", который координирует все действия. Это может облегчить отладку и мониторинг.
2. **Согласованность:** Если необходимо строгое соблюдение порядка выполнения или гарантии ACID, оркестрация может быть более подходящим выбором.
3. **Комплексные сценарии:** Если у вас есть сложные условия или зависимости между различными шагами рабочего процесса, централизованная оркестрация может сделать систему более управляемой.

Хореография:

1. **Децентрализация:** В хореографии каждый компонент системы знает, что ему нужно делать. Это может упростить масштабирование и уменьшить точки отказа.
2. **Простота:** Хореографии часто проще реализовать и понять, если у вас есть простой рабочий процесс без сложных зависимостей.

3. **Гибкость:** Хореография может быть более адаптивной к изменениям, так как каждый участник может изменять свое поведение независимо.

Нюансы при использовании паттерна

- **Откат операций:** Один из критических аспектов Saga — это компенсирующие транзакции, которые нужны для отката изменений в случае ошибки.
- **Сложность координации:** В оркестрированной саге у вас есть централизованный сервис, который должен знать о всех шагах, что может привести к сильной связанности. В хореографированной саге сложность заключается в том, чтобы отслеживать, какие события взаимодействуют между собой, что может стать сложным при росте системы.
- **Согласованность:** Saga не обеспечивает строгую ACID-согласованность, как в монолитных системах, и приходится часто иметь дело с "eventual consistency".
- **Сложность отладки и мониторинга:** Особенно в хореографированных сагах может быть сложно понять, в каком состоянии находится Saga и какие шаги уже были выполнены.
- **Порядок событий:** В хореографированных сагах порядок событий может иметь значение, и его нужно тщательно учитывать.

В контексте нашего интернет-магазина, Saga помогает синхронизировать резервирование товара, создание заказа и отправку уведомлений так, чтобы либо все операции были успешно выполнены, либо система вернулась в согласованное состояние.

- **Проблема 2: Сервис управления заказами может испытывать большую нагрузку, особенно во время распродаж.**
Нам поможет паттерн Command-side Replica

Command-side Replica — это паттерн, который предлагает создание реплики(копии) данных на стороне команд (например, операций записи) для ускорения операций чтения и разгрузки основной базы данных. Эта реплика обычно синхронизируется асинхронно с основной базой данных.

Как это реализуется в монолите

В монолитных системах репликацию часто можно настроить в рамках одной базы данных. Монолит может использовать мастер-реплику или другие схемы репликации для обеспечения более высокой производительности и отказоустойчивости.

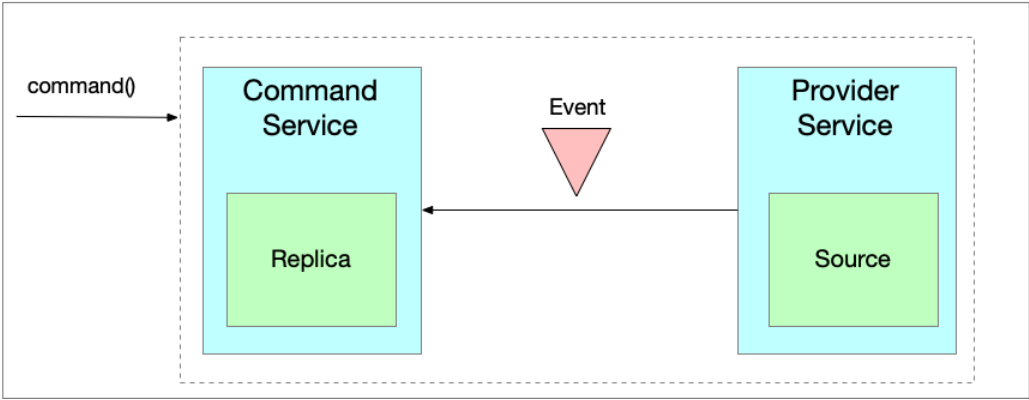
Зачем этот паттерн в микросервисах

В микросервисах, каждый сервис обычно имеет свою базу данных, и этот паттерн может помочь в разделении нагрузки между базами. Command-side Replica обеспечивает высокую доступность и производительность, делая систему более отзывчивой и надежной.

Как он реализуется

1. Сервис управления заказами в интернет-магазине имеет основную базу данных, где происходят все операции записи, такие как создание нового заказа.
2. Создается Command-side Replica, реплика(копия) этой базы, которая асинхронно синхронизируется с основной базой.
3. При большой нагрузке, например, во время распродаж, операции чтения, такие как проверка статуса заказа, направляются на эту реплику. Это разгружает основную базу данных, позволяя ей сосредоточиться на операциях записи.

Command-side replica



Нюансы при использовании паттерна

- **Актуальность данных:** Из-за асинхронной синхронизации данных между основной базой и репликой, существует риск небольших задержек в отображении актуальных данных.

- **Сложность управления:** Настройка и поддержка реплики требуют дополнительных усилий, особенно в распределенной системе.
- **Расход ресурсов:** Репликация увеличивает затраты на хранение данных и на синхронизацию между основной базой и репликой.

В контексте интернет-магазина, Command-side Replica может быть использован для разгрузки сервиса управления заказами, особенно в пиковые моменты, такие как распродажи. Операции чтения, например, поиск по заказам или их отслеживание, будут направлены на реплику, что позволит основной базе данных сосредоточиться на создании и обработке новых заказов.

- **Проблема 3: Мы хотим, чтобы другие части системы (или даже внешние системы) знали о событиях, таких как успешное создание заказа. Нам поможет паттерн Domain Event (Доменное Событие)**

Это паттерн, согласно которому система публикует события, когда происходят определенные изменения в основных сущностях. Эти события могут быть подхвачены другими сервисами для последующей обработки, что позволяет разделять сложную логику и зоны ответственности между микросервисами.

Как это реализуется в монолите

В монолитных приложениях компоненты могут подписываться на интересующие их события и реагировать на них соответствующим образом.

Зачем этот паттерн в микросервисах

В микросервисной архитектуре, доменные события особенно полезны для обеспечения низкой связанности и высокой сплочённости между сервисами. Они позволяют сервисам реагировать на действия, происходящие в других сервисах, без необходимости прямого взаимодействия между ними.

Как он реализуется

1. Клиент завершает процесс покупки в интернет-магазине.
2. Сервис управления заказами после успешного создания заказа публикует доменное событие "Заказ создан".

3. Сервис уведомлений подписан на этот тип событий и, получив его, отправляет уведомление на почту клиента о успешном оформлении заказа.

Нюансы при использовании паттерна

- **Отказоустойчивость:** Нужно уделять внимание тому, как система будет вести себя при сбоях. Что произойдет, если сервис уведомлений не сможет обработать событие?
- **Порядок событий:** В асинхронной системе может быть важным порядок событий. Это нужно учитывать при проектировании.
- **Дублирование событий:** Система должна быть устойчива к обработке дубликатов событий, что может быть актуально в распределенных системах.
- **Согласованность данных:** Поскольку события могут быть асинхронными, может возникнуть проблема согласованности данных между сервисами.

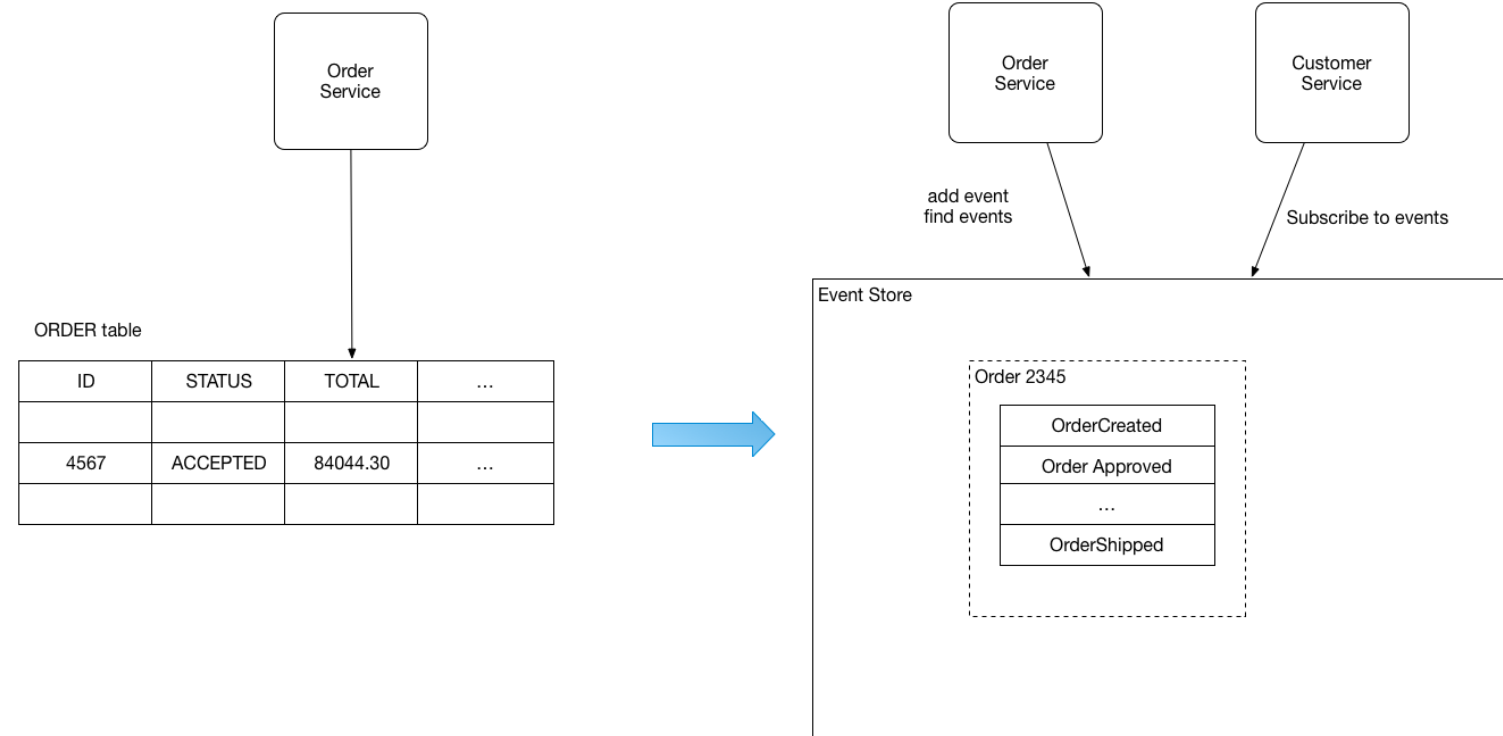
В контексте интернет-магазина, использование Domain Event позволит системе легко расширять функциональность. Например, если позже появится необходимость в ведении статистики заказов, можно просто создать ещё один сервис, который будет подписан на события создания заказов, и без изменения логики уже существующих сервисов начнет собирать необходимую информацию.

В контексте микросервисов, чтобы реализовать паттерн Domain Event, обычно используют два паттерна: Event Sourcing (Хранение событий) и Transactional Outbox (Транзакционный ящик).

1. Event Sourcing

Реализация:

В нашем примере, при создании заказа вместо прямой записи состояния заказа в свою базу данных, сервис управления заказами создаёт событие "Заказ создан" и сохраняет его в отдельном хранилище событий (куда потом по аналогии сохраняет все события, которые происходят с сущностями). Сервис уведомлений же прослушивает хранилище событий (это может быть NoSQL база или Kafka и т.д.) и при наличии события "Заказ создан" отправляет уведомление клиенту.



Плюсы:

- **Аудит и трассировка:** Поскольку все изменения сохраняются, вы можете видеть всю историю изменений.
- **Гибкость:** Легко восстанавливать и адаптировать состояние. Например, можно проанализировать историю событий и перестроить состояние.
- **Высокая производительность записи:** Поскольку только добавляются новые записи, операции записи могут быть оптимизированы.

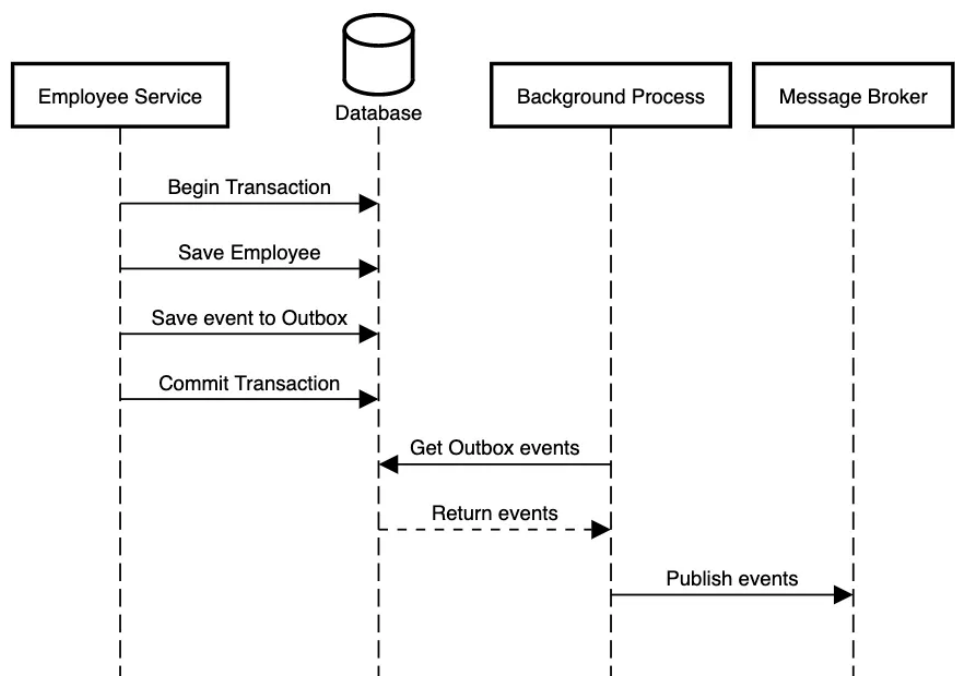
Минусы:

- **Сложность:** Этот подход требует специализированного хранилища или механизма и может стать сложным в поддержке.
- **Производительность чтения:** Чтение может требовать восстановления состояния из событий, что может быть менее производительным, чем традиционное чтение из базы.

2. Transactional Outbox

Реализация:

Когда сущность изменяется, создается запись в "ящике" (специальной таблице, которая лежит в базе данных сервиса рядом с основной таблицей). Эта запись представляет событие, которое нужно опубликовать. Отдельный процесс сервиса затем читает эти события из ящика(этой отдельной таблицы) и публикует их, например в Kafka. В нашем примере, когда заказ создается, сервис управления заказами также помещает событие "Заказ создан" в этот "ящик". А потом отдельно публикует в Kafka, где уже сервис уведомлений вычитывает это событие и отправляет уведомление клиенту.



Плюсы:

- **Согласованность:** Поскольку запись в ящик и изменение сущности происходят в одной транзакции, гарантируется, что система будет в согласованном состоянии.
- **Простота чтения:** Нет необходимости восстанавливать состояние из событий, как в Event Sourcing.

Минусы:

- **Дополнительная сложность:** Необходим отдельный процесс для чтения из ящика и публикации событий.
- **Задержка:** Может существовать небольшая задержка между созданием события и его публикацией.

Что лучше выбрать?

Выбор между Event Sourcing и Transactional Outbox зависит от конкретных требований и контекста вашей системы.

- Если вы хотите иметь возможность восстанавливать состояние объекта на любой момент времени и требуете полной аудиторской трассировки, Event Sourcing может быть хорошим выбором.
- Если ваш основной приоритет — согласованность данных и вы хотите минимизировать задержку между созданием и обработкой событий, то Transactional Outbox может быть более подходящим решением.